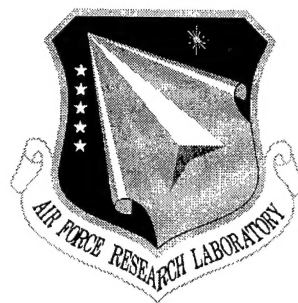


AFRL-IF-RS-TR-2001-289

Final Technical Report

January 2002



INVESTIGATION OF SOFTWARE ENVIRONMENT FOR CONFIGURABLE AEROSPACE COMMAND AND CONTROL (CACC) SYSTEMS

Syracuse University

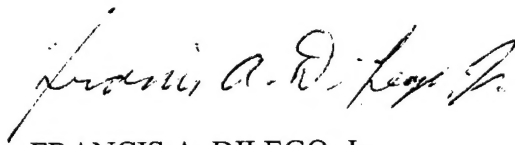
APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

20020308 055

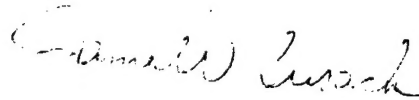
**AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE
ROME RESEARCH SITE
ROME, NEW YORK**

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

AFRL-IF-RS-TR-2001-289 has been reviewed and is approved for publication.



APPROVED: FRANCIS A. DILEGO, Jr.
Project Engineer



FOR THE DIRECTOR: JAMES W. CUSACK
Chief, Information Systems Division
Information Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory Rome Research Site mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFSA, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188	
<small>Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.</small>				
1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE Jan 02	3. REPORT TYPE AND DATES COVERED Final Jun 99 - Jan 01		
4. TITLE AND SUBTITLE INVESTIGATION OF SOFTWARE ENVIRONMENT FOR CONFIGURABLE AEROSPACE COMMAND AND CONTROL (CACC) SYSTEMS		5. FUNDING NUMBERS C - F30602-99-2-0508 PE - 62702F PR - 558S TA - CA WU - PO		
6. AUTHOR(S) C.Y. Roger Chen				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Syracuse University Office of Sponsored Programs 113 Bowne Hall Syracuse, NY 13244-1200		8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFRL/IFSA 525 Brooks Road Rome, NY 13441-4505		10. SPONSORING/MONITORING AGENCY REPORT NUMBER AFRL-IF-RS-TR-2001-289		
11. SUPPLEMENTARY NOTES AFRL Project Engineer: Francis A. DiLego, Jr., IFSA, 315-330-3683				
12a. DISTRIBUTION AVAILABILITY STATEMENT Approved for public release; distribution unlimited.		12b. DISTRIBUTION CODE		
13. ABSTRACT (Maximum 200 words) To overcome large scale system design problems conventional approaches that consist of: 1) system functionality definition, 2) partitioning of the system functionality into sub-functional blocks, 3) independent development of each sub-functional block into a sub-system component, and 4) connection of sub-system components to form an entire system, one has to address the foundation of a Configurable Aerospace Command Center (CACC) system, such that it will provide the capability of allowing sub-systems and components to be pluggable into the environment and form a coherent system, where exchanges of events, data and functions across the boundaries of sub-systems will be seamless and actions on all sub-systems will be globally coordinated and optimally controlled. In this report, we investigate many such issues and look into existing systems, and perform feasibility assessment on existing collaboration systems to see how much intersystem collaboration functions can possibly be developed through an external functional extension, i.e., without the access or modification of the source code of existing collaboration systems.				
14. SUBJECT TERMS Java, Component Object Frameworks Object Oriented Design		15. NUMBER OF PAGES 76		
		16. PRICE CODE		
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL	

Chapter 1 Table of Contents

1. Requirements of Configurable Aerospace Command and Control (CACC) Systems	1
2. A Natural Software Architecture for CACC – Component-Based Development (CBD).....	3
3. Problems in Component-Composing Techniques.....	5
4. Composing CCBS out of COTS Components.....	6
4.1. Component Specifications and Interaction Specifications	7
4.2. The Nature of Components	9
4.3. The Properties of Components	10
4.4. Component Interactions	14
4.5. Requirements for Composing Components	18
4.6. Implicit Entities in an Interaction Specification	20
4.6.1. Connector.....	20
4.6.2. Component Container.....	22
4.6.3. Instantiation Control	23
4.6.4. Inter-framework Communicator.....	25
4.6.5. Sequencing Control	26
4.6.6. Data-model Translator	28
4.6.7. Communication Pattern Handler	30
4.6.8. Anomaly Observer.....	31
(1) UML as an ADL	33
5. Intelligent Software Synthesis System.....	41
6. KOSOVA WAR ASSESSMENT PRESENTATION – A Conceptual Demonstration for Feasibility.....	45

6.1.	Overall Look of the Application.....	46
6.2.	Connection to CVW for Collaboration.....	53
References.....		62

Figure 1: The steps of constructing a component-based application

Figure 2: Meta-model of Software Architecture

Figure 3: The Microsoft Component Object Model (COM)

Figure 4: JavaBeans Application

Figure 5: ORBA ORB Architecture

Figure 6: The COTS Component-based Architecture

Figure 7: The “4+1” View Model

Figure 8: UML Constructs used in Sequence Diagram

Figure 9: System Diagram for the Kosova War Demonstration

Figure 10: A Demonstration Snapshot

Figure 11: Map Viewer Component

Figure 12: Image Viewer Component

Figure 13: Video Display Component

Figure 14: PowerPoint Display Component

Figure 15: Web Display Component

Figure 16: Use case of Kosova Demonstration

Figure 17: The Static Structure of Kosova Demonstration

Figure 18: The Behavior Specification of Kosova App.

Figure 19: The Static Structure of a Presentation Component

Figure 20: The behavior specification of the presentation in figure 3 (part 1)

Figure 21: The behavior specification of the presentation in figure 3 (part 2)

Figure 22: The Static Structure of the CVW Connection

Figure 23: The behavior Specification of the CVW Connection (part 1)

Figure 24: The behavior Structure of the CVW Connection (part 2)

1. Requirements of Configurable Aerospace Command and Control (CACC) Systems

The development of a CACC involves a wide range of advanced technologies including object modeling, databases, distributed computing, communications, networks, multimedia, visualization, and collaboration. In addition, the nature of CACC imposes a unique requirement of mobility, which implies that the characteristics of the location, where a CACC is deployed, vary, especially in terms of networking capability, traffic loads/congestion, and functional requirements. Thus, the ability of system reconfigurability and self-adjusting, based on local specifics, becomes very critical.

The increasing complexity of knowledge/information storage, communications, processing and visualization as well as system/network infrastructures has made the conventional approaches to developing such information environments very inefficient.

Conventional approaches in designing such a large-scale system involve (1) system functionality definition, (2) partitioning of the system functionality into sub-functional blocks, (3) independent development of each sub-functional block into a sub-system component, and (4) connection of sub-system components to form an entire system. Problems with such an approach when used to develop a CACC is the lack of coordination between the development of individual sub-system components, as well as the lack of global planning in the foundation of a system.

More specifically, potential drawbacks/risks using conventional approaches to designing a CACC are described in the following:

- Minimum interoperability between sub-systems or components
- Very Limited system scalability and expandability
- Inability to self-adjust
- System performance degradation
- Long design turnaround time (or time-to-field)
- Error-proneness in system management

To overcome such potential problems, one has to address the foundation of a CACC system, such that it will provide the capability of allowing sub-systems and components to be pluggable into the environment and form a coherent system, where exchanges of events, data, and functions across the boundaries of sub-systems will be seamless and actions on all sub-systems will be globally coordinated and optimally controlled.

In this proposed effort, we will investigate many such issues and will look into existing systems, and perform feasibility assessment on existing collaboration systems to see how much inter-system collaboration functions can possibly be developed through an external functional extension, i.e., without the access or modification of the source code of existing collaboration systems.

2. A Natural Software Architecture for CACC – Component-Based Development (CBD)

Component-based software development (CBD) is the latest solution for software problems. The principle of CBD is “Software Reuse.” Commercially-Off-The-Shelf (COTS) components perfectly comply with this principle. Users of the COTS components do not modify the source in any way, since the vender of a COTS component provides only the interfaces that define the behavior of this COTS component. In short, COTS components are the ideal building blocks for a component-based software system. In order to construct COTS component-based software systems (CCBS), developers need a new type of tool to perform the process of composing these COTS components into a software system.

Software developers use software architectures (SA) to suppress implementation detail of component-based applications, and to concentrate on the analyses and decisions that satisfy requirements of these applications.

Architecture description languages (ADLs) provide notations for developers to describe software architectures. Computer Aid Software Engineering (CASE) tools help developers transforming these notations into code of a certain programming language or executable realizations of a certain platform.

On the other hand, software developers expect component-based applications to be collections of components, rather than “Components + Scripts.” Therefore, component-based software architectures are “Components + Interactions”, rather than

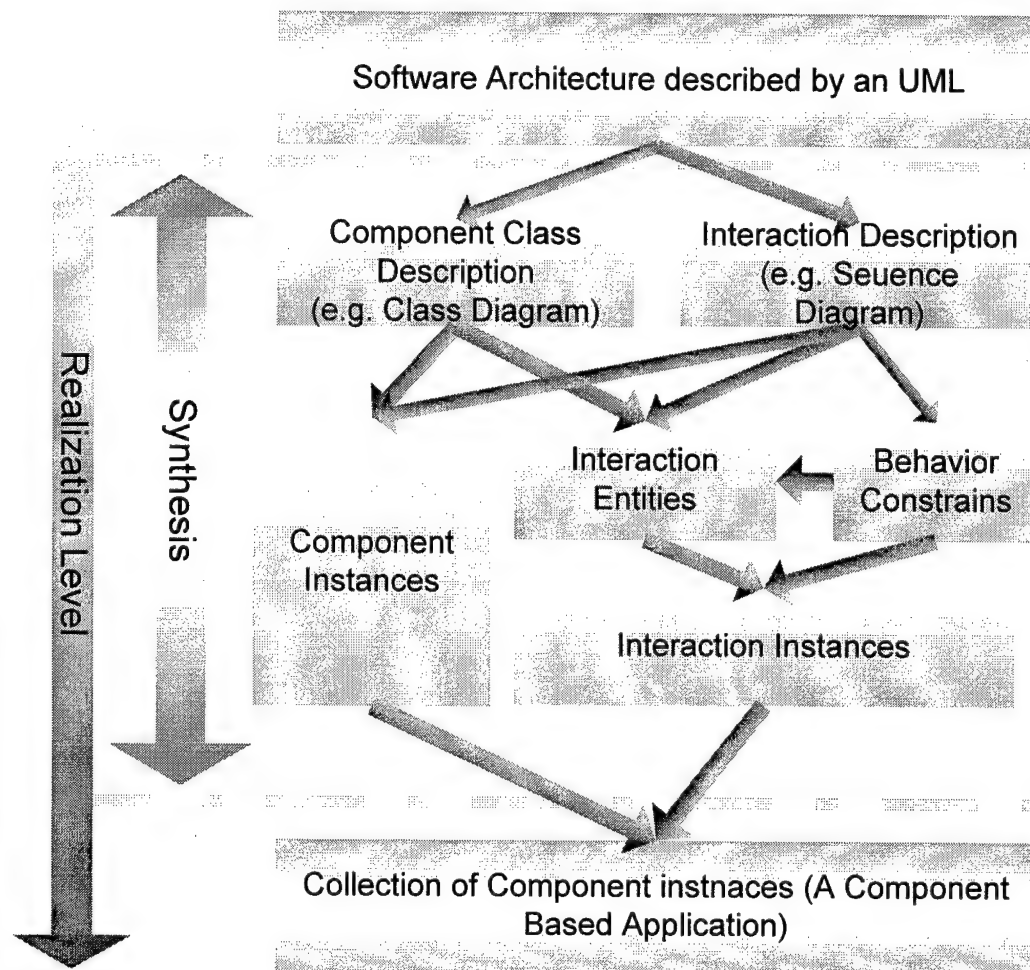
“Components + Connectors + Behaviors Constraints,” where scripts are used to describe behavior constraints. In other words, software architectures are “Components + Interactions”; and component-based applications are collections of components.

Unified Modeling Language (UML) is the standard notation for modeling software systems. UML class diagrams and sequence diagrams provide enough notations for software developers to describe component specifications and interaction specifications for component-based software architectures.

With these foundations, a new type of component-based software synthesis tool should be able to:

- (1) Transforms the “interaction specification” in the description of component-based software architecture to “component instances”, which carries out these “requirements of interactions.”
- (2) Realizes the specification of component instances in the architecture using UML.
- (3) Generates the component composition of that architecture for a specific executable model.

This component-based software synthesis tool helps application developers build component-based systems out of commercial-off-the-shelf (COTS) components. Such a design flow is shown in Figure 1.



1.1.1.1 Figure 1 The steps of constructing a component-based application

3. Problems in Component-Composing Techniques

In order to achieve CBD, researchers have proposed various component-composing techniques. However, there are three major problems in those techniques:

1. Existing component-composing techniques fail to meet the expectation of developers who want to pursue the component-based approach. Few component-composing techniques compose applications using software components with neither glue code nor application-defined plug-in mechanisms.
2. Little discussion focuses on “reusing” the existing standard modeling language to model the component-based applications other than re-inventing “composition languages.”
3. Software components are “software ICs.” It is difficult to find a software synthesis system, which produces and verifies the component-based applications, resembling the ones in handling “hardware ICs.”

4. Composing CCBS out of COTS Components

Few component-composing techniques compose applications using software components with neither glue code nor application-defined plug-in mechanism. This problem occurs at the design phase of component-based software development process. In general, software architecture provides solid basis for large-scale development of distributed applications that use COTS components implemented in multiple programming languages [1]. In other words, software architecture offers an ideal abstraction for supporting the development of CCBS.

Recently published papers often favor the view that software architectures are “components + connectors + behavioral constraints.” The components in the context of

software architecture description are the abstractions of the component instances in a software system. Therefore, this view leads to that component-based applications are “components + scripts”, which dominates the current component-composing techniques. However, component integration also addresses two issues: (1) packaging components so that they can be connected at run-time, and (2) connecting, disconnecting and re-connecting components at run-time [2]. It is very difficult to handle these two issues with modeling component-based applications as “components + scripts”, because the scripts define the control flow at design-time. This observation motivates the search for more suitable definition of component-based applications and software architecture. The alternative views regarding “software architecture” and “component-based applications” are:

- Software architectures are “Components + Interactions.”
- Component-based applications are collections of components.

4.1. Component Specifications and Interaction Specifications

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A component represents a computational, functional, or data unit in a software system.

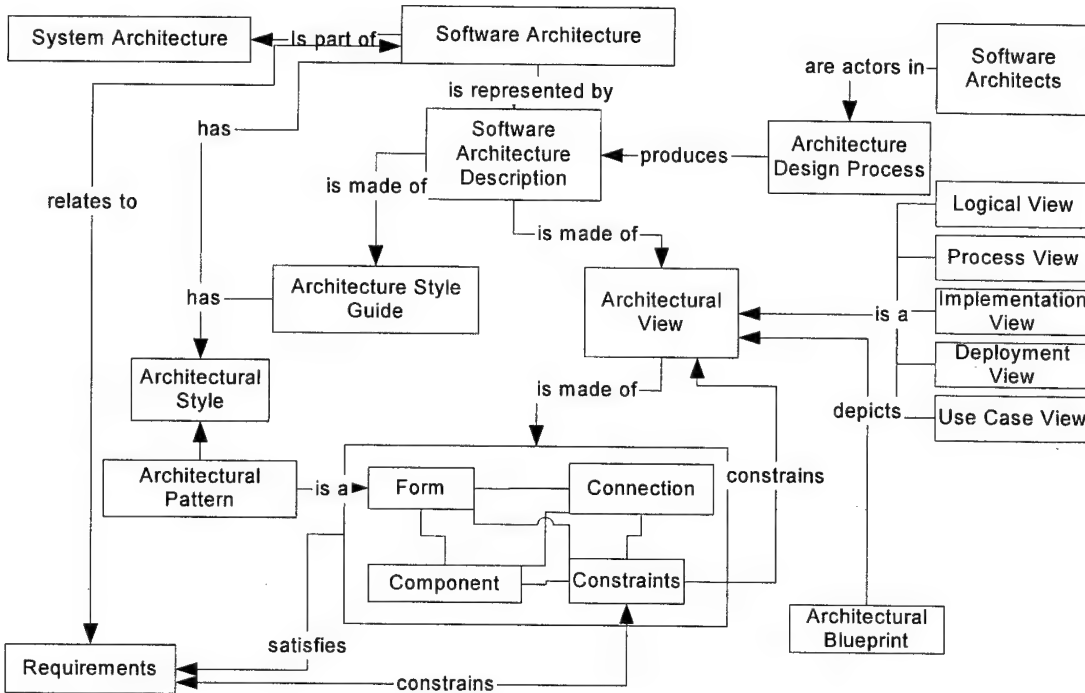
In a software system, an interaction represents the connections between components, including data flow and control flow (execution, invocation), as well as interaction

constraints, including component communication protocols, visibility, timing, and synchrony.

A component specification, which is not modifiable, of a software architecture defines the functional constraints of components. It is the static aspects of a software architecture, components are specified that have to be filled in order to perform the task of this software system.

The dynamic part of a software architecture is described by interaction specifications.

An interaction specification represents constraints of the communications among components, including constraints of system configurations. In brief, it is the combination of the constraints of all the architectural elements, which are components and interactions, forms behavioral constraints of a software system. The meta-model of a software architecture is shown in Figure 2.



1.1.1.2 Figure 2 Meta-model of Software Architecture

4.2. The Nature of Components

A component specification can abstract a function, data, package, cluster and system abstraction or a system structure [3]. The nature of components is classified as:

1. Design Components: A component can be a design principle or idea.
2. Specification Component: A specification can be considered a reusable component. Specifying the expected functionality and behavior of a component frees the developer to implement this component in a variety of programming languages [4].

3. Executable Components: The source code of those components is not available, and the executable components themselves are commercially available (COTS).

4.3. The Properties of Components

The component properties are defined as follows:

- **Component Class:** A component class describes the structure and behavior of a single component class or a collaboration of classes.
- **Component Family:** Although a software component is nearly independent of other components, it rarely stands alone. A group of components designed for collaborating with each other forms a “component family.”
- **Component Framework:** A software component assumes one or more specific architectural context. This architectural context is the “component framework” of this software component. A component framework is a collection of software components with a software architecture that determines the interfaces that components may have and the rules, which govern their composition.
- **Component Functionality:** A software component fulfills a clear function. A software component is logically and physically cohesive, and denotes a meaningful structural and behavioral chunk of a software system.
- **Component Interfaces:** A software component conforms to a set of interfaces. The interface specifies the services (messages, operations, and variables) that a component provides. When a software component conforms to a given interface it means that this

software component satisfies the contract specified by that interface, and may be substituted in any context wherein that interface applies.

- **Substitutability:** A software component is a replaceable piece of a system. A software component is substitutable for another component, which realizes the same interfaces. It helps the evolution of a software system, once deployed by making it possible to upgrade and evolve parts of the system independently.
- **Hierarchical Construction:** A software component exists in this component framework and represents a fundamental building block of a software system. Furthermore, this construction is hierarchical: a software system at one level of abstraction is a software component at a higher level of abstraction.
- **Introspection:** The specification of interfaces of a software component can be discovered at run-time.
- **Component Repository:** Every component framework provides a component repository engine that ships as a component. A COTS component installation program must register the shipped component classes in one of component repositories.

These properties of a software component illustrate that a software component is a conceptual unity of design, and construction.

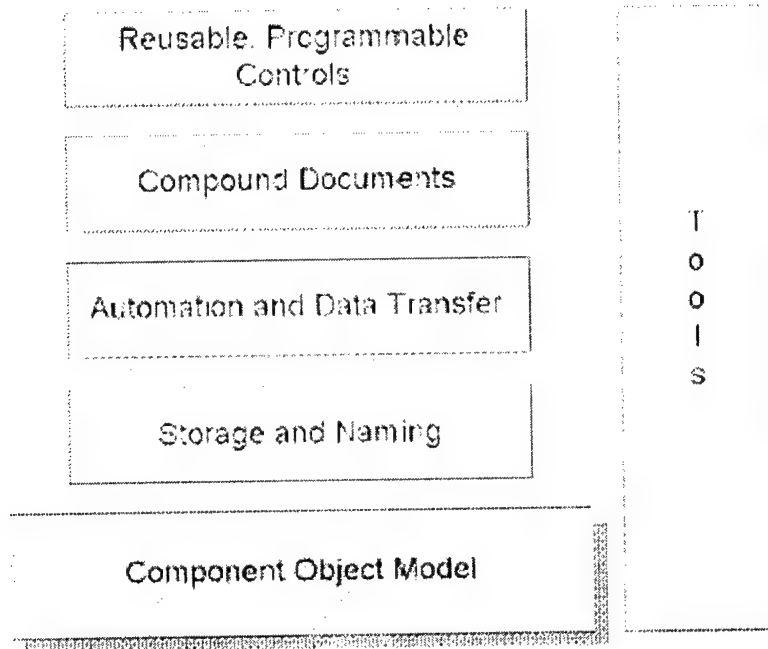


Figure 3 The Microsoft Component Object Model (COM)

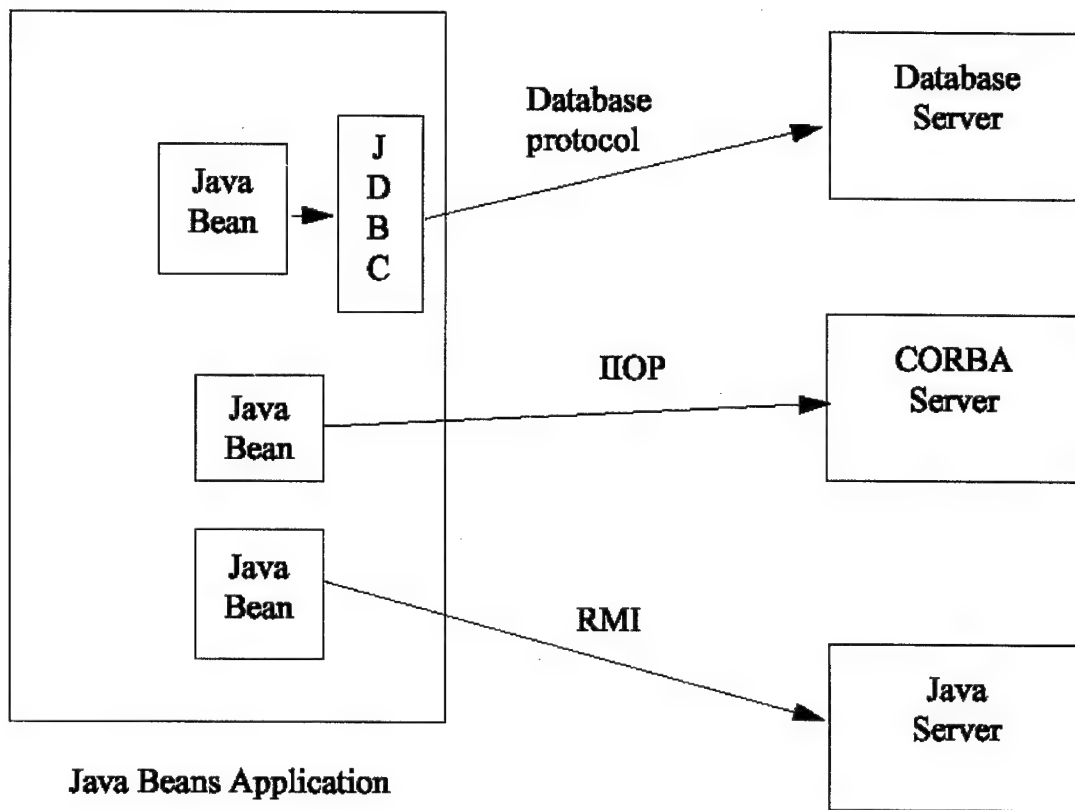


Figure 4 JavaBeans Application

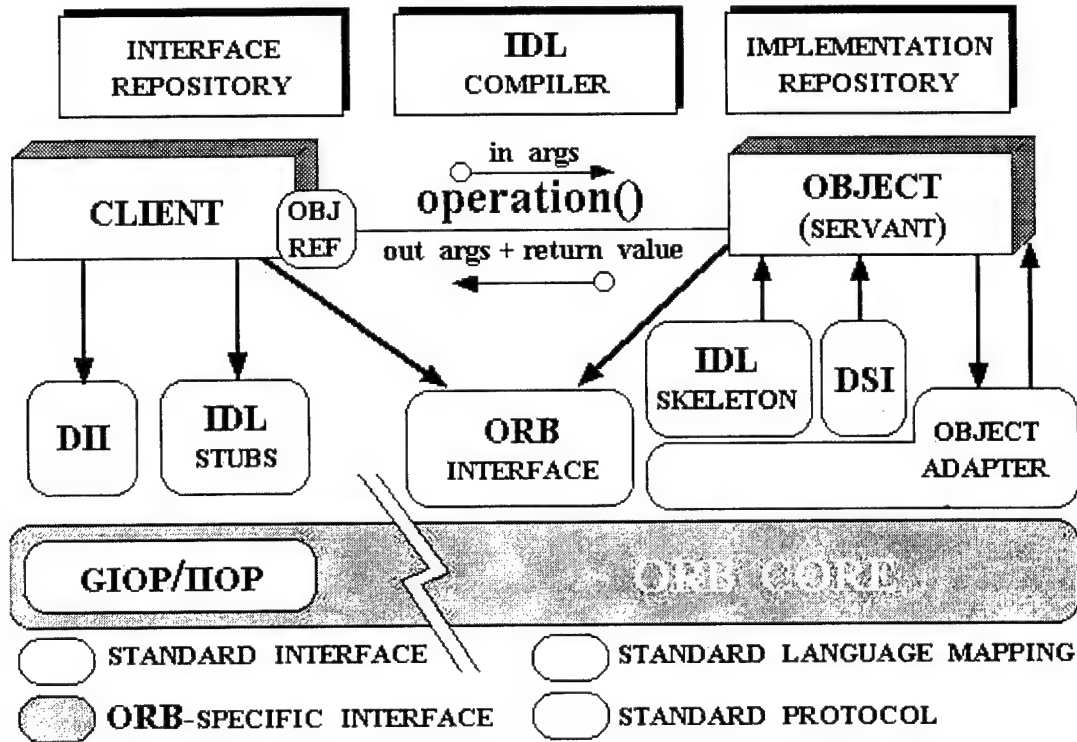


Figure 5 ORBA ORB Architecture

4.4. Component Interactions

The dynamic part of a software architecture is described by interactions. An interaction declares how and in which order stimuli complying to its message are to be exchanged. Integration problems arise when a software component depends on certain assumptions concerning its interactions with its environment, but is to be placed into a software system that is based on different assumptions. The result is interaction protocol mismatches. There are four types of interactions and their possible mismatches with the environment:

1. **Component-platform interactions.** A software component must be executed somewhere. It can be either a real processor and an operating system for binary executables, or a virtual one. If a software component was compiled for one type of platform, it will need an emulator or a code converter in order to run it on another platform.
2. **Component-hardware interactions.** A software component can interact directly with hardware component through a certain communication protocol, such as writing and/or reading from ports. If the port's numbers are different from what is expected by the component, the component must undergo some modification.
3. **Component-user interactions.** A component's user interface requirements may also change. For example, a component can have its messages in one language, when the system requires another language.
4. **Inter-component interactions.** A software component always interacts with other software components, and there can be mismatches between the components. A set of possible mismatches between components [5]: representation, communication, packaging, synchronization, semantics, control, etc.

These four types of interaction protocol mismatches must be overcome and investigated in order to reuse COTS components. This report focuses on the study of inter-component interactions. There are four main categories for causing the mismatches in inter-component interactions.

1. **Assumptions about the nature of the components:** Within this category, there are three areas: (1) infrastructure—assumptions about the substrate on which the component is built; (2) control model—assumptions about which component(s) (if any) control overall the sequencing of computations; (3) data model—assumptions about the way the environment will manipulate data managed by a component.
2. **Assumptions about the nature of the connectors:** Within this category, there are two areas: (1) protocols—assumptions about the patterns of interaction characterized by a connector; (2) data model—assumptions about the kind of data that is communicated.
3. **Assumptions about the global architectural structure:** These include assumptions about the topology of the system communications and about the presence or absence of particular components and connectors.
4. **Assumptions about the construction process:** In many cases, the components and connectors are produced by instantiating a generic building block. For example, a database is instantiated, in part, by providing a schema; an event broadcast mechanism is instantiated, in part, by providing a set of events and registrations. In such cases, the building blocks frequently make assumptions about the order in which pieces are instantiated and combined in an overall system.

In order to compromise these possible mismatches, the realization of an interaction specification must possess the following entities:

1. **Inter-framework Communicator:** It communicates with more than one component frameworks if required. If two or more involved components of an interaction specification assume different component frameworks, the corresponding realization of this interaction must contain a service or a group of services that can communicate with all the specified component frameworks. This capability eliminates the possibility of having mismatches caused by different assumptions about the substrate on which the component is built.
2. **Sequencing Control:** This guarantees a certain sequence of computation if required. This capability provides the overall sequencing control mechanism in order to guarantee a certain sequence of computation specified in the software architecture.
3. **Data-model Transformation:** It provides a suitable data model for each involving component.
4. **Communication Pattern Handler:** It identifies the communication pattern required by communicating components. The realization of an interaction must handle either synchronous or asynchronous communication patterns.
5. **Anomaly Observer:** It is connected to external monitoring services to observe the anomaly of a software system if such services are available and are required.
6. **Instantiation Control:** It ensures the order of the instantiation of components. This requires the realization of an interaction specification to provide the overall instantiation control mechanism in order to ensure the order of the instantiation of components specified in a software architecture.

7. **Connector:** It receives one or more registered messages from senders and invokes the corresponding action according to the interaction specification. This capability is required because the communicating components may assume different interfaces for communication. This is always the case for independently developed components.
8. **Component Container:** It provides identifications for components in a software architecture. A component container instantiates the contained components according the specification given by an instantiation control. A component container also defines the necessary interfaces and rules, which the contained components must comply.

4.5. Requirements for Composing Components

It has been specified [6] that the following characteristics are necessary to compose components together: (1) specifying connections of the components, (2) bridging architectural styles, (3) adapting components that have not been designed to work together, and (4) managing dependencies between concurrent and distributed components. These characteristics indicate the following technical requirements:

1. **Communication and binding:** components' behavior consists in the exchange of messages; components' functionality is instantiated by binding parameters to values; components may themselves be values.
2. **Concurrency:** an application is a concurrent composition of objects whether or not there may be multiple concurrent threads active at any time.

3. Choice: a component typically provides an interface consisting of a choice of services.
4. Abstraction: components are abstract entities whose behavior and functionality is only accessible through their interface;
5. Instantiation: components can be dynamically instantiated and thus it must be possible to generate new names for components and their communication channels, and to communicate these names with existing components.

The following is an explanation of each technical requirement and its corresponding capabilities and properties provided by the realization of an interaction and component specification:

1. Communication and binding: components' behavior consists in the exchange of messages that is handled by **connectors** according to the **component interfaces** defined by each involved **component class**; **components' functionality** is instantiated, by **instantiation control**, by binding parameters to values; components may themselves be values. The specification of interfaces of a software component can be discovered at run-time. (Introspection)
2. Concurrency: an application is a concurrent composition of objects whether or not there may be multiple concurrent threads active at any time. **Sequencing Control** guarantees the message sequence of each thread; **Communication Pattern Handler** handles either synchronous or asynchronous communication patterns. **Anomaly Observer** usually is required for resolving resource conflict.

3. Choice: a software component typically provides one interface of **component interfaces** consisting of a choice of services. The specification of interfaces of a software component can be discovered at run-time. (Introspection)
4. Abstraction: components are abstract entities whose behavior and **functionality** is only accessible through their **interface**;
5. Instantiation: components can be dynamically instantiated (**Instantiation Control**), so it must be possible to generate new names, by the **Component Container**, for components and their communication channels, and to communicate these names to existing components.

4.6. Implicit Entities in an Interaction Specification

A realization of an interaction specification must provide the following entities, which are connector, component container, inter-framework communicator, sequencing control, communication pattern handler, data-model translator, instantiation control, and anomaly observer. Whether these entities are abstract or not depends on the given software architecture, and necessity based on the regarding components and component frameworks. However, all these entities do offer precise semantics to a software system.

4.6.1. Connector

A connector receives one or more registered messages from senders and invokes the corresponding action according to the interaction specification. A connector provides

service for the communicating components, which assume the different interfaces for communication. This is always the case for independently developed components.

Connectors mediate interactions but are not “things” to be hooked up. Each connector has a protocol specification that defines its properties. These properties include rules about the types of interfaces it is able to mediate for assurances about properties of the interaction, rules about the order in which things happen, and commitments about the interaction such as ordering, performance, etc. Each is of some type or subtype (e.g., remote procedure call, pipeline, broadcast, event). The specific named entities visible in the protocol of a connector are roles to be satisfied (e.g., client, server).

To avoid confusion with the traditional “connector,” in this project we use “I3S connector” as the identification of this entity. An I3S connector is a component class that consists of following attributes, methods, and notifications. Furthermore, an I3S connector accepts multiple messages that come from different senders. This section just illustrates the basic properties, which handle the connection among communication components, of an I3S connector. The following sections may add some other properties into the I3S connector.

1. Attributes of an I3S connector:

- Sources: The names of component instances that send messages to a target component instance and the names of messages that are sent to a target component instance.

- Target: The name of a component instance that receives messages from source component instances and the name of a method that is expected to be invoked.
 - Parameter Mapping: The mapping between the parameters of messages of source component classes and the parameters of the method of the target component class.
2. Methods of an I3S connector:
 - Reset: Clear all the assignments to this I3S connector.
 3. Notifications generated by an I3S connector:
 - Forward Message: Forward each received message.
 - Invocation Succeed: Indicate the intended invocation has been successfully performed, and the return value is the argument of this notification.
 - Invocation Failed: Indicate the intended invocation has failed.

4.6.2. Component Container

Similar to an STL container template class, a component container manages a set of component instances, which can be instances of any component class that complies with the requirements of residing component framework as well as the requirements implied by a component container. This section describes the basic properties required of a component container. A special component container class may have additional parameters, and additional methods.

A component container must define the necessary interfaces and rules, which the contained components must comply. Therefore, an I3S component container is a component class that consists of following attributes, methods, and notifications.

1. Attributes of an I3S component container:

- Iteration model: The method of retrieving a component.
- Supported component frameworks: The component frameworks that the container supports.
- Instantiation Sequence: The sequence of how the container instantiates container components.

2. Methods of an I3S component container:

- Add: Adds a component to the container.
- Get A Component: Retrieves a component in the container.
- Remove: Removes a component from the container.

3. Notifications generated by an I3S component container:

- Insanitation Succeed: Indicates that the intended insanitation sequence has been successfully performed.
- Insanitation Failed: Indicates that the intended insanitation sequence has failed to conduct.

4.6.3. Instantiation Control

Instantiation control assures the order of the instantiation of component instances. This requires the realization of an interaction specification to provide the overall instantiation

control mechanism in order to assure the order of the instantiation of components specified in a software architecture.

When an identification of the component class is known, the development tool needs to include the identification of the component class when saving the instantiation sequence of components persistently. This persistent component is the **instantiation control**. The instantiation control, which is saved in the format required by the component container, describes the exact source of each component's class code, which is referenced at run time, in addition to each component's instance initialization data. An instantiation control provides necessary information for a component container to create previously saved instances of component classes with some persistent data existing in the instantiation control. For every component framework, instance data has an associated identification of component class—thus, a component container knows which class code it has to instantiate. Once the component container creates that instance, it initializes the instance with persistent data saved in the instantiation control. The storage format of an instantiation control depends on the target component container and the component framework.

The instantiation control is a component class that consists of following essential attributes, and methods.

1. Attributes of an instantiation control:

- SupportedComponentContainer: The supported component container.

2. Methods of an instantiation control:

- Read: Reads data from the instantiation control.
- Write: Writes data to the instantiation control.

4.6.4. Inter-framework Communicator

Integrating a software component into a software system is significantly easier when this software system consists of software components that are built to the same component framework. There are many COTS component frameworks including CORBA, COM/ActiveX, JavaBeans, Java class libraries, and Microsoft Visual Basic controls. However, if two or more communicating components of an interaction specification assume different component frameworks, the corresponding realization of this interaction must contain mechanism that can communicate with all the specified component frameworks. In recognition of this need, products whose function is to bridge between components written to different frameworks arise, like the CORBA/COM bridge and the ActiveX/JavaBean bridge. Therefore, component frameworks do not need to consider a component's inter-framework communication when attempting to reuse it. Furthermore, it is indicated [7] that no single component frameworks will ever be appropriate for all software components.

This section captures following attributes, methods, and notifications of an inter-framework communicator.

1. Attributes of an Inter-framework Communicator:

- Sources: The names of component instances that send messages to a target component instance; and the names of messages that are sent to a target component instance.
 - Target (Optional): The name of a component instance that receives messages from source component instances and the name of a method that is expected to be invoked.
 - Parameter Mapping: The mapping between the parameters of messages of source component classes and the parameters of the method of the target component class.
 - Supported Component frameworks.
2. Methods of an Inter-framework Communicator:
- Reset: Clears all the assignments to Inter-framework Communicator.
 - Merge: Merges with other inter-framework communicators that support different component frameworks.
3. Notifications generated by an Inter-framework Communicator:
- Forward Message: Forwards each received message.
 - Invocation Succeed: Indicates that the intended invocation has been successfully performed, and the returned value is the argument of this notification.
 - Invocation Failed: Indicate the intended invocation has failed.

4.6.5. Sequencing Control

A sequencing control provides the overall sequencing control mechanism in order to guarantee a certain sequence of computation specified in the software architecture. A sequencing control is the mechanisms for managing the flow of control among the components. It shows the major execution sequences, where execution sequences may be asynchronous or parallel, and how synchronization is managed. It also allows anomalous conditions such as error handling and exception conditions that may dynamically alter the flow of execution. To guarantee a certain sequence of specified computations, a sequencing control exists as I3S connector's properties.

When a message is sent to an I3S connector, this I3S connector must validate that input messages are received according to the specified sequence. A software architect may specify the number of occurrences of an interaction and the time elapse between sequenced messages. In this case, a counter and a timer are required for an I3S connector. Furthermore, a sequence of messages may be unconstrained, but there may exist combining conditions among messages, such as AND constraints, and OR constraints. In this case, an I3S connector synchronizes these messages following the combining conditions. Therefore, an I3S connector must have the following attributes, methods, and notifications to perform the sequencing control.

1. Sequencing Attributes of an I3S connector:

- Sequence Mapping: The mapping between the source messages and the receiving order and/or time elapse.
- Counter: The number of occurrences of the specified method invocation.

- Combining conditions: The combining conditions among source messages.
2. Sequencing Methods of an I3S connector:
 - Reset Sequence: Resets the sequence to the initial state.
 - Reset Counter: Resets the counter.
 - Unconditional invocation: Invokes the target method regardless of the sequencing control.
 3. Sequencing notifications generated by an I3S connector:
 - Time Expired: Exceed the specified time elapse for a sequenced message.
 - Out of sequence: Indicate the arrival of an out-of-sequence message.

4.6.6. Data-model Translator

The data-model translator provides the suitable data model for each involving component. Data modeling, as it applies to component-based software systems, is the process of defining the vocabulary and content to be used to represent information in a component-based software system. The vocabulary defines the terminology to be used to describe, or attribute, individual data elements; and the content defines what data is to be included in the system and what is not. The modeling process is independent of specific hardware and software, which only become important at a subsequent implementation stage. To be useful, the data modeling process must be carried out in an open, inclusive manner, so that the community of eventual users has adequate input into the design process. The whole reason for the data modeling process is to build systems that make information more useable and effective [8].

Each component defines its own logical construct for the storage and retrieval of information. This logical construct consists of a collection of data structures, a collection of operators, and a collection of integrity rules. A data model defines the elements required to describe those aspects of the 'real world', which it is designed to model, and the nature of the links between these elements. This is achieved by rigorous analysis of the elements and the ways in which they interact when this component is designed. To avoid ambiguity, the data model itself employs very precise definitions of the terminology it uses. The COTS component framework, which this component resides, is responsible of these definitions. Similar to a programming language, a COTS component defines the basic data types and their ranges, and the rules of converting these data types to each other. Therefore, one of data-model translator's tasks is applying these rules into the communication in case there is a data-model mismatch caused by basic data type mismatch. However, this solution only applies to the communicating components residing in the same component framework, and the mismatch is due to basic data type mismatch.

Two approaches were used to resolve the data-model mismatch. For the data-model mismatches occurring between components residing in the same component framework and the mismatch being caused by component framework defined primitive data types, the I3S connector handling the communication of involving components implicitly calls the default **data type converter** to resolve this mismatch. For complex data types conversion and inter-framework data-model mismatch, the development tools

automatically constructs an XML document for each relating data entity of communicating components, and interactively request decisions from the software developer until the mismatch has been resolved. Furthermore, each data-model translation or complex data type conversion forms a new data converter and stores as an **I3S data converter**. Therefore, an I3S connector must have following attributes, and notifications to resolve the data-model mismatch.

1. Data conversion related attributes of an I3S connector:
 - Data model mapping: The mapping between the data model mismatches and the I3S data converters.
2. Data conversion related notifications, generated by an I3S connector:
 - Data conversion failure: The process of converting data has failed.

4.6.7. Communication Pattern Handler

A Communication Pattern Handler identifies the communication pattern required by communicating components. The realization of an interaction must handle either synchronous or asynchronous communications. Communications are synchronous when the sender component of a message must wait for a response from the receiver component of the message before performing subsequent tasks. The period that the sender component must wait depends on how long it takes the receiver component to handle the message and produce a response. On the other hand, with asynchronous communications, a sender component sends a message to a receiver component and moves on to subsequent tasks immediately. If a response from the receiver component is

expected, the sender component decides when it actually looks for and processes the response. However, there is no guarantee that a receiver component will process the message within any particular period.

A Communication Pattern Handler is an implicit entity that keeps track of the incoming messages' communication patterns for an I3S connector. Therefore, an I3S connector must have following attributes, and notifications to handle different communication patterns.

1. Communication related attributes of an I3S connector:

- Communication pattern mapping: The mapping between the source messages and their communication patterns.
- Time constraints: The time constraints for the synchronous messages.

2. Communication related notifications generated by an I3S connector:

- Communication Time Expired: Exceeds the specified time constraint elapse for a synchronous message.

4.6.8. Anomaly Observer

An Anomaly Observer is an entity that traps and reports the anomalous behaviors of a software system. A component-based software development introduces additional sources of risk because (i) independently developed components cannot be fully trusted to conform to their published specifications and (ii) very often, software failures are caused by systemic patterns of interaction that cannot be localized to any individual

component. A software failure is the inability of a software system to fulfill a task it was intended. An associated term is “fault” that is the actual cause of the failure. This leads to the fault management that aims at handling faults before they cause a failure. Fault management has three phases: monitoring to detect fault symptoms, diagnosis to identify the type and location of the fault and correction to eliminate the fault [9]. Therefore, a separate exception-handling infrastructure is necessary to address these issues. An approach is proposed [10] to separate the exception handling functions from normative functions in component-based software systems. Components focus on executing their own “normal” problem solving behavior, while an exception handling service focuses on detecting and resolving exceptions in the system as a whole.

In highly complex systems, it is unrealistic to expect that automated processes can completely detect, diagnose, and resolve all possible exceptions. Furthermore, the fault diagnosis, resolution, and correction are usually off-line tasks. Even an automated system can resolve or correct some minor faults during the run-time; the underlying causes still exist in the design of some particular components, or the overall system configurations or designs. The most important task is to report these failures in order to help component developers and system architects find an optimal solution for the observed run-time faults. Therefore, the task for an Anomaly Observer assigned by this article is to detect fault symptoms and identify the location through the failure notifications from the I3S connectors. The diagnosis and fault correction is beyond the scope of this project.

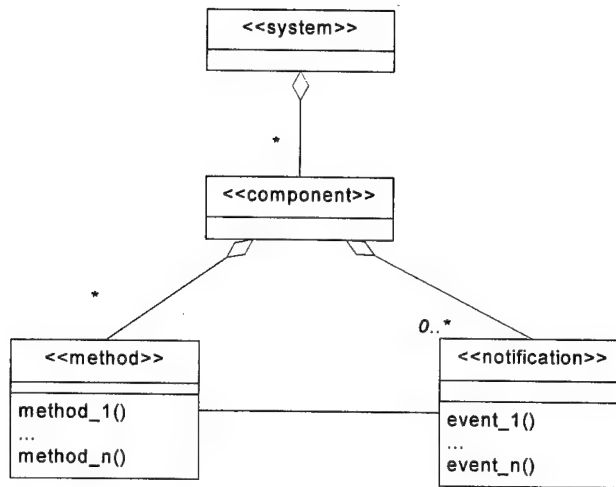


Figure 6 The COTS Component-based Architecture

(1) UML as an ADL

This section presents a solution for solving the second problem: Little discussion focuses on “reusing” the existing standard modeling language to model the component-based applications other than re-inventing “composition languages.”

UML stands for Unified Modeling Language. UML is an object modeling technique that evolved because of the combined work of James Rumbaugh, Grady Booch, and Ivar Jacobson; each of them had their own OO modeling notation. The Object Management Group (OMG) adopted UML as a standard for software modeling in late 1997. UML is now the standard for software modeling. UML provides a blueprint for developers so they know exactly what they need to build and for project managers so that they can precisely estimate the cost of a given project. UML is able to bridge between technical

developers and their non-technical users through a set of interrelated diagrams of the various parts of a software system. Moreover, UML allows the developers to get a precise understanding of the exact requirements that the users have for the system being built.

The software architecture of a component-based software system defines its high-level structure as a collection of communicating components. To avoid the deficiencies of using ad-hoc and informal notations to describe architecture, the software engineering research community has pioneered ADLs that have well-defined semantics and tools for parsing, compiling etc. However, instead of using ADLs, industry has focused on the UML. Many companies are using UML for architectural description. In fact, the UML developers are advocating this use and the introduction to the latest version states: “One of the key motivations in the minds of the UML developers was to create a set of semantics and notation that adequately addresses all scales of architectural complexity, across all domains [11].”

The shortcomings of UML as an ADL are discussed in the following:

1. **The formal analysis and semantics of software architecture.** Within the UML semantics, OCL is used as invariants on the metaclasses in the abstract syntax as well as to define ‘additional’ operations. In other words, OCL is used as syntactic notation for component constraints, but not as notation to define an appropriate semantics for a software system. However, component constraints need to be considered at the same level as the component semantics. OCL is also too implementation-oriented and not well suited for conceptual modeling [12] and

formal analysis. The UML does provide a definition of the static format of the model. This meta-model is considered as a semi-formal definition since a meta-model is expressed in UML class diagrams. Improvements in these semi-formal UML semantic descriptions are needed to convey a rigorous semantic representation and provide tool support to verify UML diagrams against an unambiguous specification of UML semantics [13]. A likely alternative is to introduce a formal language, which specifies component semantics and additional properties for constraining components.

2. An effective tool for the direct execution of constructed UML models, analyzing the software architectures, active specification, and supporting dynamism. Model checking is a method for formally verifying finite-state concurrent systems. Symbolic algorithms are used to traverse the model defined by the system and check if the specification holds or not. Neither Microsoft Visio nor Rational Rose is a model checker. Analysis of architectures is rather primitive in either Microsoft Visio or Rational Rose, since the complete formal semantic model of UML is yet to be finalized. Both Microsoft Visio and Rational Rose provide limited support for active specification. For Rational Rose, it is a programming language based support, which might involve the syntax checking for desired programming languages. The Microsoft Visio emphasizes the drawing function rather than semantics checking or analyzing. However, both provide some connection points for importing customized functionality that enables sophisticated support for active specification. UML itself does not support run-time evolution, or dynamism. The

limited support for modeling dynamism in existing ADLs, including UML, is reflected in the limited tool support for dynamism. Neither Rational Rose nor Microsoft Visio provides the desired manner of supporting dynamisms similar to MILs: Architectural components are implemented in a programming language and the architectural description serves only to ensure proper interconnection and communication among them.

3. **Automatic application generation.** UML are used as modeling notations and many UML CASE tools, such as Rational Rose for J [14], C++ [15], and Visual Basic [16], provide implementation generation support. However, they do not take the final refinement step from architectural descriptions to source code. For example, Rational Rose for C++ [15] provides a C++ class hierarchy for its concepts and operations. This hierarchy forms a basis from which an implementation of a software architecture is produced. Application skeletons produced by Rational Rose for C++ facilities result in instantiated, but partially implemented, framework classes.
4. **Description of interfaces as first class entities.** The general interface description can be found in the UML component diagrams, which are not intended to represent the logical decomposition of a software system into reusable and combinable subsystems. For the interface definition, UML uses a simple list of the interface methods. This information is put inside the class definition in a UML class diagram. The class diagrams also show the relations among modules in addition to a complete description of the modules.

5. **Description of connectors, which specify component inter-connections precisely and intuitively, as first class entities.** UML does not offer the concept of connectors as first-order objects, which would be a hybrid of an association (association class) and a dependency between a class and an interface of another class.
6. Hierarchical architectures for supporting software evolution. UML itself does not support run-time evolution, or dynamism.
7. **Description of stylistic constraints.** This shortcoming of UML is due to the lack of formal notation of describing architectural constraints.

The solutions for shortcomings of UML being an ADL are:

1. **The formal analysis and semantics of software architecture.** UML's OCL is used as syntactic notation for component constraints, but not as notation to define an appropriate semantics for a software system. On the other hand, component constraints need to be considered at the same level as the component semantics. OCL is also too implementation-oriented and not well suited for conceptual modeling [12] and formal analysis. The UML provides a definition of the static format using meta-model that is expressed in UML class diagrams. A class diagram is typically not refined enough to provide all the relevant aspects of a specification [11]. To avoid ambiguities introduced by natural language, formal languages have been developed. However, formal languages are only usable to people with a string mathematical background, but difficult for the average business or system modeler to use [11]. Therefore, the task of formalizing a software system should be

automated, and supported by a tool. A formal semantics for a modeling notation can be obtained by defining a meaning function from syntactic structures in the UML diagrams to artifacts in the formally defined semantic domain. This section defines meaning functions for UML class diagrams and sequence diagrams in order to justify the correctness of the modeled systems. We use the PROMELA [17] as the formal notation for system specifications.

2. An effective tool for the direct execution of constructed UML models, analyzing the software architectures, active specification, and supporting dynamism. Model checking is a method for formally verifying finite-state concurrent systems. Model checking is usually done by the direct execution of a constructed software architecture, i.e. a UML model, before the final executable image has been generated. There are several model checkers for UML, such as SPIN, vUML, xUML, cTLA, etc. We use an existing model checker with the I3S.
3. **Automatic application generation.** The I3S generates executable systems based on the given UML models.
4. **Description of interfaces as first class entities.** The general interface description can be found in the UML component diagrams. For the interface definition, UML uses a simple list of the interface methods. This information is put inside the class definition in a UML class diagram. This can be viewed as a user friendliness issue. Although there is no conclusion that which way is better, it is a fact that UML is much more popular than any modeling notation that “is” an ADL.

5. **Description of connectors, which specify component inter-connections**

precisely and intuitively, as first class entities. UML does not offer the concept of connectors as first-order objects, which would be a hybrid of an association (association class) and a dependency between a class and an interface of another class. However, we do not consider this is an important issue for synthesizing COTS component based applications. Furthermore, EDCS, that sponsors these research on ADLs such as C2, Wright, CHIME, and ACME, states clearly that (software) architecture describes component topology and interactions in terms of legal and illegal configurations and sequences of events [18].

6. **Hierarchical architectures for supporting software evolution.** UML itself does not support run-time evolution, or dynamism. We do not intend to increase or modify the notation of UML. Instead, we use the I3S to overcome this shortcoming. However, UML does support hierarchical construction in which UML treats the model of a system as a component.
7. **Description of stylistic constraints.** This shortcoming of UML is due to the lack of formal notation of describing architectural constraints. The translation of stylistic constraints of a software system is beyond the scope of this project. However, with proper meaningful functions, the stylistic constraints can be translated into a desired formal language, and verified by the associated tools.

A UML model captures the static and dynamic features of a software system. A software system modeled by UML notation uses various diagrams for describing the static and dynamic features of this system. This UML model, which consists of a set of interrelated

diagrams, often starts from a typical set of views, which is the 4+1 view model as shown in Figure 7.

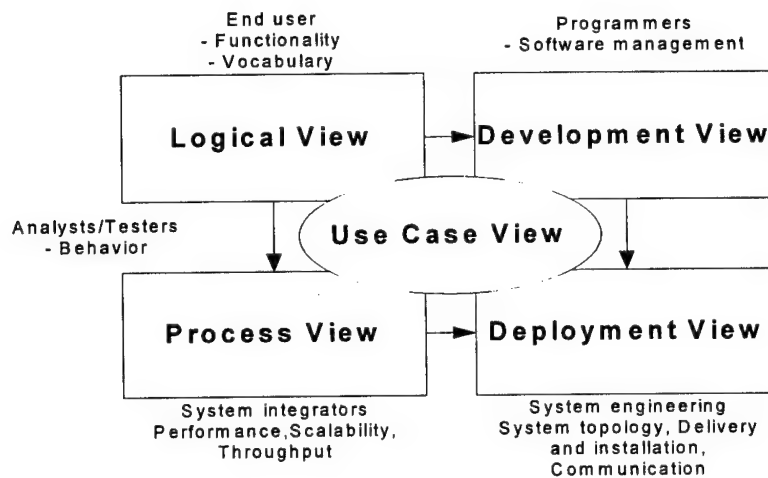


Figure 7 The “4+1” View Model

However, different diagrammatic representations of the same information may vary in the computational efficiency of working with these diagrams [19]. In order to efficiently work with UML diagrams and define meaning functions for the essential UML diagrams, next section explores the effects of diagrammatic representations on the task of integrating multiple diagrams focusing on modeling COTS component-based systems.

There are three major obstacles for accomplishing this goal of using UML actually developing COTS component-based systems. To overcome these obstacles, we propose the following solutions:

1. UML can function as an ADL. This section summarizes this discussion, and uses a classification and comparison framework for software ADL to classify UML. In this section we also propose a solution for effectively using UML as an ADL.
2. UML uses various diagrams for describing system behavior. However, different diagrammatic representations of the same information may vary in the computational efficiency of working with these diagrams. In this section we explore the effects of diagrammatic representations on the task of integrating multiple diagrams focusing on modeling COTS component-based systems.
3. The existing UML CASE tool is not facilitated for synthesizing COTS component-based systems. This issue will be discussed in next section.

In addition to these issues, we formalize the class diagrams and sequence diagrams with focus on modeling COTS component based software systems in this section.

5. Intelligent Software Synthesis System

In This section we propose a solution for solving the third problem: Software components are “software ICs.” It is difficult to find a software synthesis system, which produces and verifies the component-based applications, resembling the ones in handling “hardware ICs.” To solve this problem, we use I3S that helps application developers build component-based systems out of commercial-off-the-shelf (COTS) components. I3S directly maps the design specifications using the notations of UML class diagrams and sequence diagrams at instance level to an executable realization that is a collection of

COTS components without the tedious work of gluing or wrapping components together using particular texture programming or scripting language.

We will describe the synthesis steps for transforming the interactions among component instances into attributes of instances of the interaction entities. We use the SPIN to model-check the result of this transformation. The previous section indicates that the existing UML CASE tool is not facilitated for synthesizing COTS component-based systems. Therefore, I3S emphasizes the solving of this shortcoming; I3S is implemented as an add-on component to an existing UML tool.

One of the important tasks to an effective technique for software reuse is the automatic mapping from abstraction specification to abstraction realization [1]. Moreover, Software components are “software ICs.” However, It is difficult to find a software synthesis system, which produces and verifies the component-based applications, resembling the ones in handling “hardware ICs.”

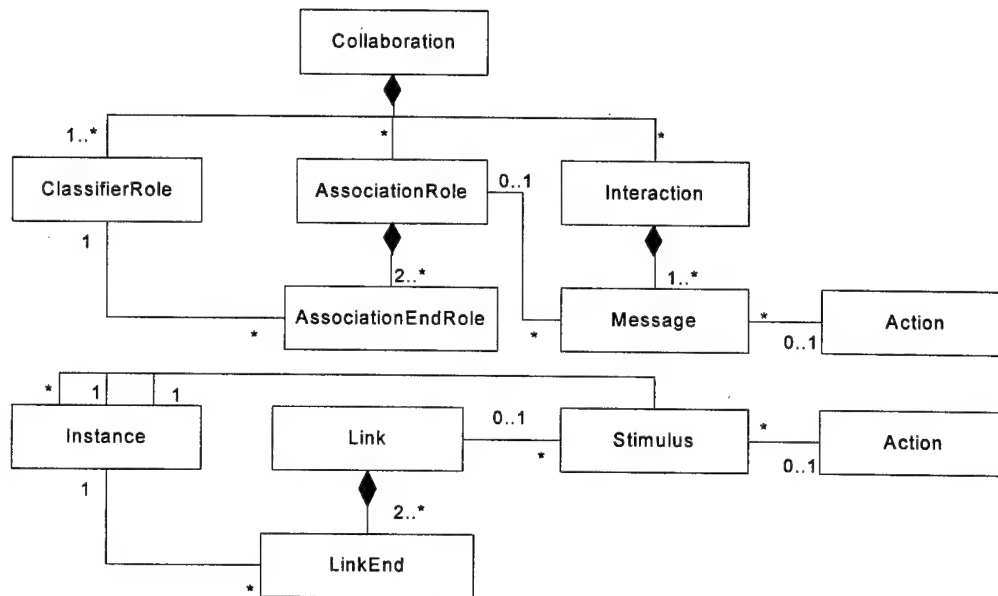


Figure 8 UML Constructs used in Sequence Diagram

A synthesis tool targeting at the component instance level consists of three major steps:

Design specification: Object-oriented analysis and design methods, such as UML [11], provide a well-suited toolbox for component-based application development, but current practice shows that the OO technology hinders the development of component-based systems. Object-oriented analysis and design methods are domain-driven, which usually leads to designs based on domain objects. Most of these methods assume that developers build an application from scratch, and they incorporate reuse of existing architectures and components too late in the development process (if at all) [20]. On the other hand, component-based software development emphasizes identifying and managing

interactions among preexisting pieces of software in order to integrate them into new systems. The UML sequence diagrams provide information on identifying and managing interactions among components in a set of scenarios. A sequence diagram gives a useful global view of the possible interactions between component instances, without referring to particular executions of the system. Therefore, a sequence diagram is a synthesis of all participants in a set of scenarios. I3S uses UML sequence diagrams as the design specifications for system behavior.

Design Implementation: A sequence diagram shows an interaction arranged in time sequence. In particular, it shows the instances participating in the interaction by their “lifelines” and the stimuli they exchange arranged in time sequence. Therefore, the implementation must evaluate the performance of the precision of the time interval, the number of iterations, and the target platform of the results obtained with respect to the design specifications. The synthesis system must have the knowledge of the software components available in the design library, which is a collection of UML class diagrams in I3S. The synthesis system then selects the target containers, allocates of the software components, identifies the involving software components, and inserts the connectors. The process is similar to convert the RTL description or model to a logic level implementation.

Design Verification: UML defines the formal syntax and semantics for sequence diagrams. Besides that, a formal definition of connectors and components is necessary for verifying the composed software. I3S uses SPIN, which is a specification technique using

formal language PROMELA that is generally based on CSP, as the model checker. I3S generates the formal description of the interaction entities, component instances, and sequence diagrams as inputs for SPIN.

As a result, I3S generates the collection of components from UML sequence diagrams and class diagrams. Furthermore, with the formal syntax and semantics of sequence diagrams and the formal definition of connectors and components, I3S uses SPIN to model-check the synthesis of components and connectors.

6. KOSOVA WAR ASSESSMENT PRESENTATION – A Conceptual Demonstration for Feasibility

The Kosova War Assessment Presentation Application is a component based application. The data for the application is stored in a database. The user asks a query to the database by selecting a region on map, database answers that query and the data returned in the query result are handled by different components. The block diagram of the application is shown in Figure 9.

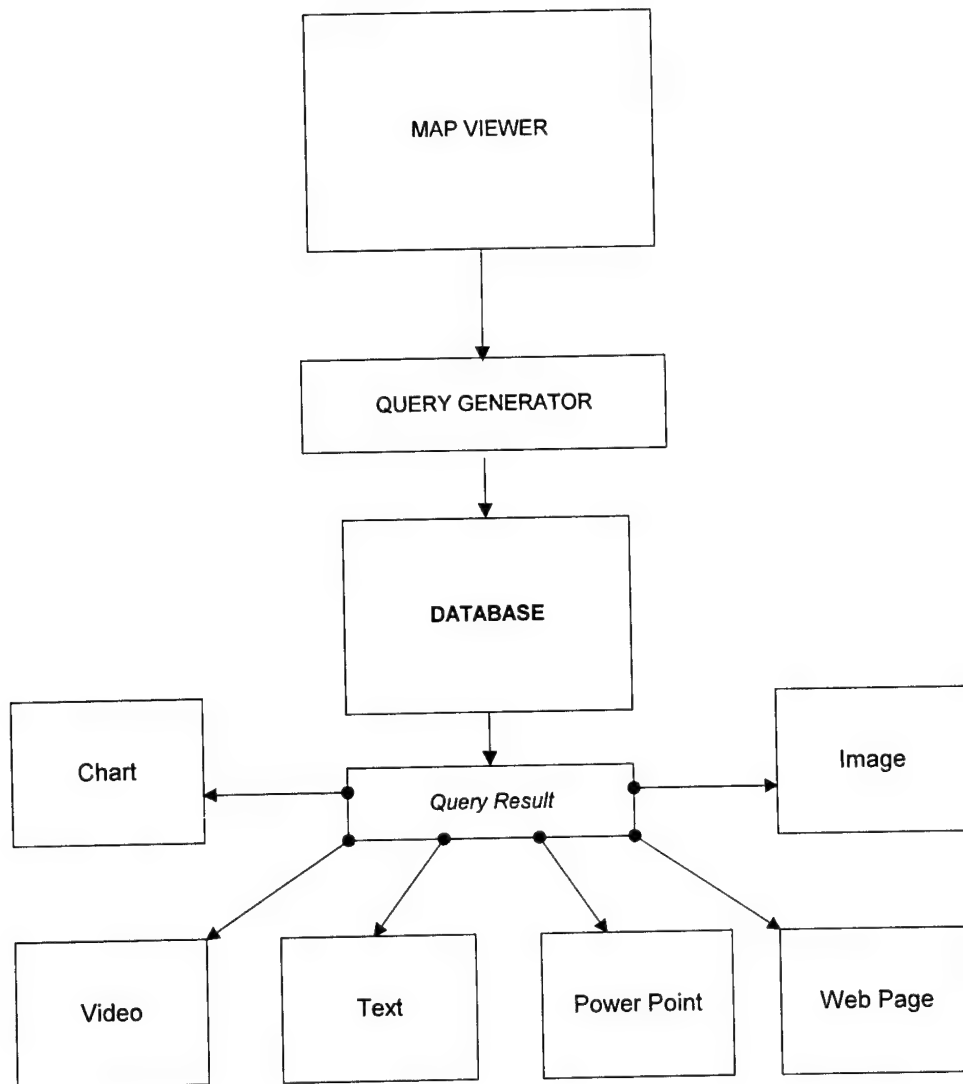


Figure 9. System Diagram for the Kosova War Demonstration.

6.1. Overall Look of the Application

The application looks like the one shown in Figure 10. The frames can be resized and displaced, and the look can be changed by the user.

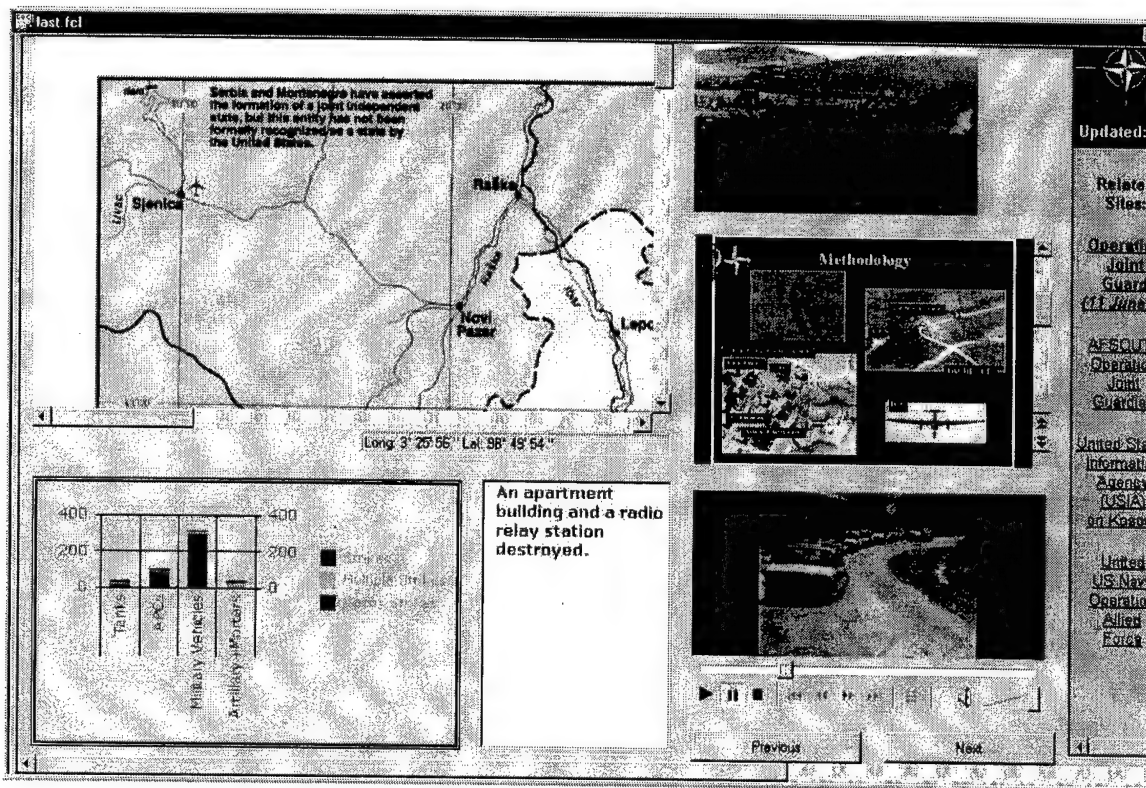


Figure 10. A Demonstration Snapshot.

First, the user selects a region on the map to see the war assessment in that region. Upon selection, a query about that region is generated and sent to the database. The database returns the results for the query and then the results are automatically displayed in the corresponding frames. A chart about the destructions in that region is displayed on a chart. An image from the region is displayed in an image frame. Two videos related to

that region are displayed in video frames. Any one of videos can be selected to be played using the buttons beneath the video frame. A short information about the region is shown in a text frame. A PowerPoint frame displays the power point presentation about that selected region. Also a web page related to that region is displayed on a web page frame. Frames can be moved or resized on the application.

City name	co x	co y	Information	Image	Video	PowerPointFile	WebSite	Video2
Pec	23	58	Ammunition Storage site	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w4.htm	c:\formctrl\
Vilaminica	22	59	Command center destro	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w5.htm	c:\formctrl\
Klina	35	56	Ammunition Storage site	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w6.htm	c:\formctrl\
Orahovac	39	46	Command center destro	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w4.htm	c:\formctrl\
Suva Reka	46	43	An airfield, TV transmits	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w7.htm	c:\formctrl\
Glogovac	49	56	Ammunition Storage site	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w1.htm	c:\formctrl\
Leposavic	45	79	Command center destro	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w9.htm	c:\formctrl\
Podujevo	61	71	An airfield, TV transmits	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w10.htm	c:\formctrl\
Movo Bido	71	55	Ammunition Storage site	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w11.htm	c:\formctrl\
Kosovska	77	54	Command center destro	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w12.htm	c:\formctrl\
Kacanik	65	37	An airfield, TV transmits	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w13.htm	c:\formctrl\
Vitina	69	41	Railway Petroleum loadi	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w1.htm	c:\formctrl\
Sibica	44	62	An apartment building a	c:\formctrl\demo\i	c:\formctrl\demo\	c:\formctrl\demo\	c:\formctrl\demo\w2.htm	c:\formctrl\

The information is stored in a database indexed by the city name in the regions.

The information is handled by components. The components used in the development are:

Map Viewer Component: This component displays single or multiple piece maps with defined coordinates. Allows user to select a rectangular region on the map and returns the coordinates of the selected region.

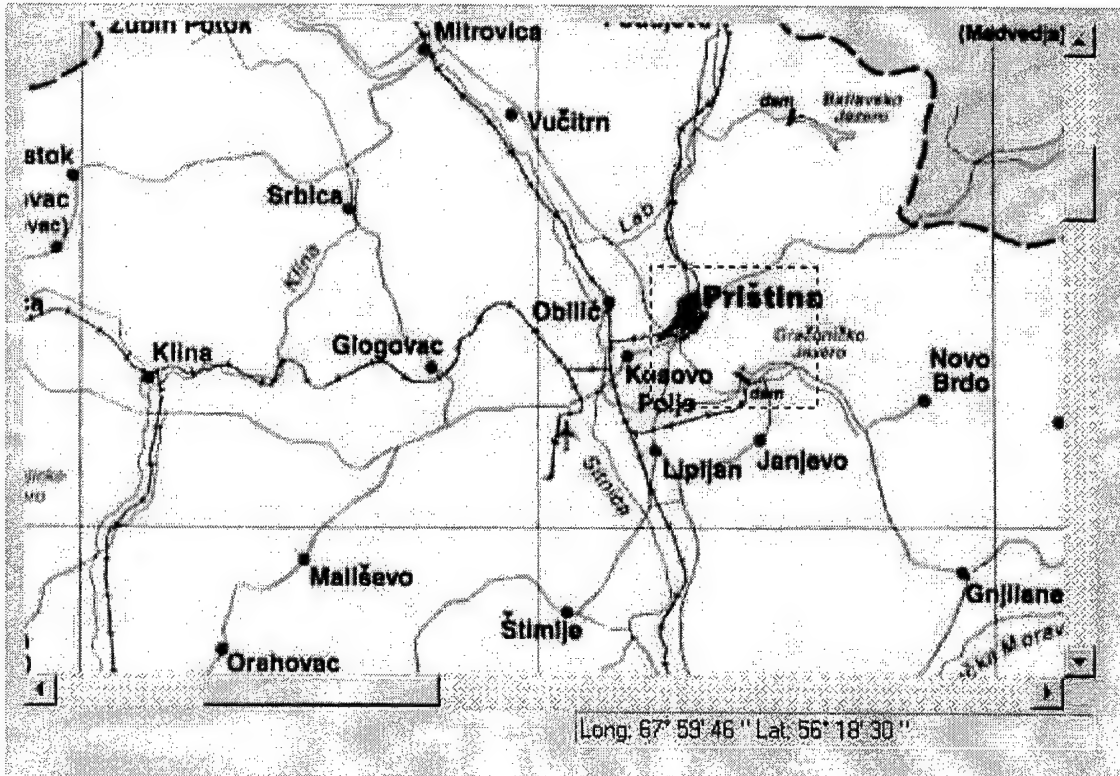


Figure 11. Map Viewer Component.

Query Generator Component: This component generates a query using the coordinates returned from map viewer and sends that query to data control to be handled.

ADO Data Access Component: This component handles all interaction with the database. It sends queries, stores returned query results, handle updates.

Chart Component: This component draws the chart of information related to destruction returned from database as the query result.

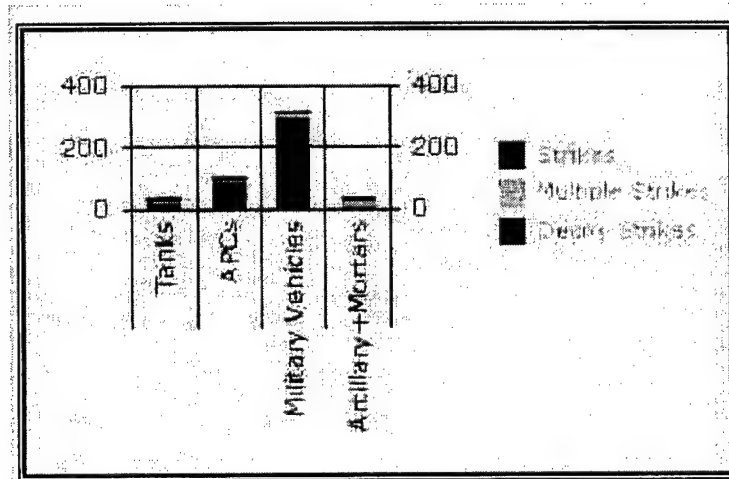
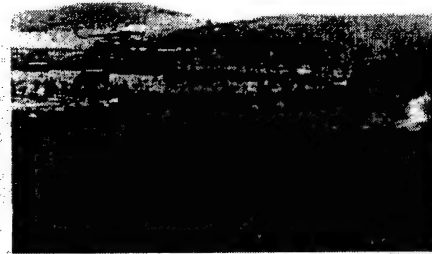


Image Viewer Component : This component is used to display the images associated with the region selected and returned by database as the query result.

Figure 12. Image Viewer Component.



Video Display Component: This component is used to display videos associated with the selected region and returned in the query result by the database. There are two videos associated with each selected region. The user can go to the next or previous video by using the buttons under the video display. Also, this component allows random access to the video frames and pausing.

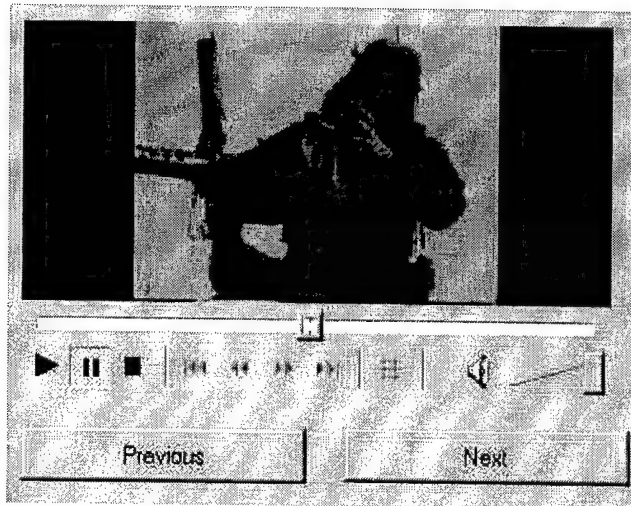


Figure 13. Video Display Component.

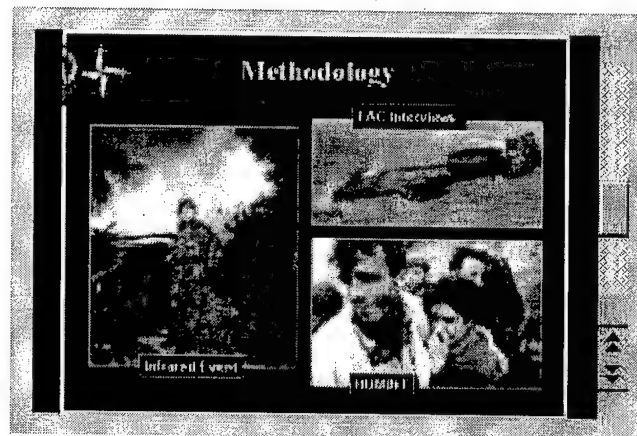


Figure 14. PowerPoint Display Component

PowerPoint Display Component: This component handles the display of PowerPoint presentations associated with the region selected. It allows navigation through the slides.

Web Display Component: This component displays the web pages associated with the selected region. It is a fully functional web browser without the buttons and menus.

(11 June)
in relation to the conflict in Kosovo.

AFSOUTH
Operation
Joint
Guardian

United States
Information
Agency
(USIA)
on Kosovo


United
US Navy
Operation
Allied
Force

Operational updates

Factual updates on Operation Allied Force were produced on a daily base at 09.00 (local time).

Mon	Tue	Wed	Thu	Fri	Sat
7 June	8 June	9 June	10 June		
31 May	1 June	2 June	3 June	4 June	5 June
24 May	25 May	26 May	27 May	28 May	29 May
17 May	18 May	19 May	20 May	21 May	22 May
10 May	11 May	12 May	13 May	14 May	15 May
		7 May	8 May(*)	9 May	

(*) indicates ACE News Briefing



Morning briefings

Daily morning briefings at NATO HQ at 11.00 (local time) on the situation in Kosovo and Operation Allied Force were given by NATO Spokesman

Mon	Tue	Wed	Thu	Fri	Sat
7 June	8 June	9 June	10 June	11 June	12 June
31 May	1 June	2 June	3 June	4 June	5 June
24 May	25 May	26 May	27 May	28 May	29 May
17 May	18 May	19 May	20 May	21 May	22 May
10 May	11 May	12 May	13 May	14 May	15 May
3 May	4 May	5 May	6 May	7 May	8 May

Figure 15. Web Display Component

6.2. Connection to CVW for Collaboration

Kosova demonstration is a system that presents interactive presentations collaboratively to audiences on the Internet. It uses a Collaborative Virtual Workspace (CVW) server, which is developed by MITRE Corporation, to provide a virtual workspace where audiences can communicate, collaborate, and share information, regardless of their geographic location. The CVW utilizes client/server architecture to implement the shared virtual space. The CVW client and Server maintain persistent connections and communicate via TCP/IP and MOO Client Protocol (MCP). MCP supports textual communication, navigating and interacting with objects in the virtual space, and receipt of state information.

It is developed using I3S that demonstrates its capability of reusing existing COTS components, composing components hierarchically, conducting inter-framework message exchange, and performing message synchronization by integrating I3S interaction entities with the selected COTS components.

The use case of the demonstration is shown Figure 16.

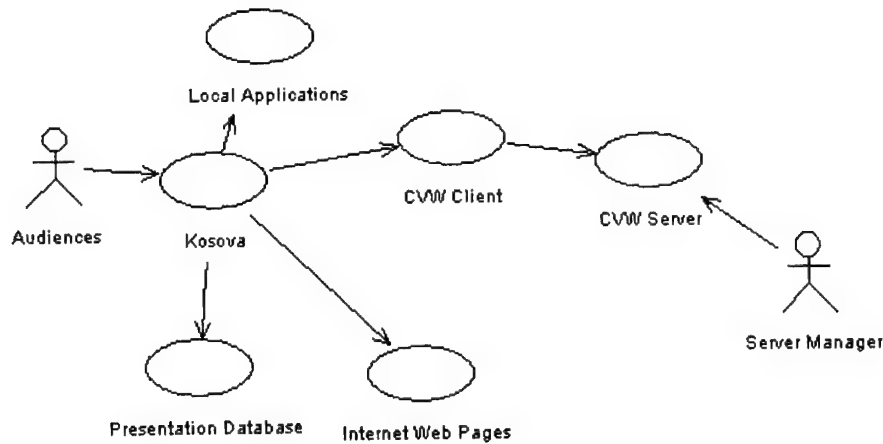


Figure 16. Use case of Kosova Demonstration.

Figure 17 shows the static structure of the Kosova application.

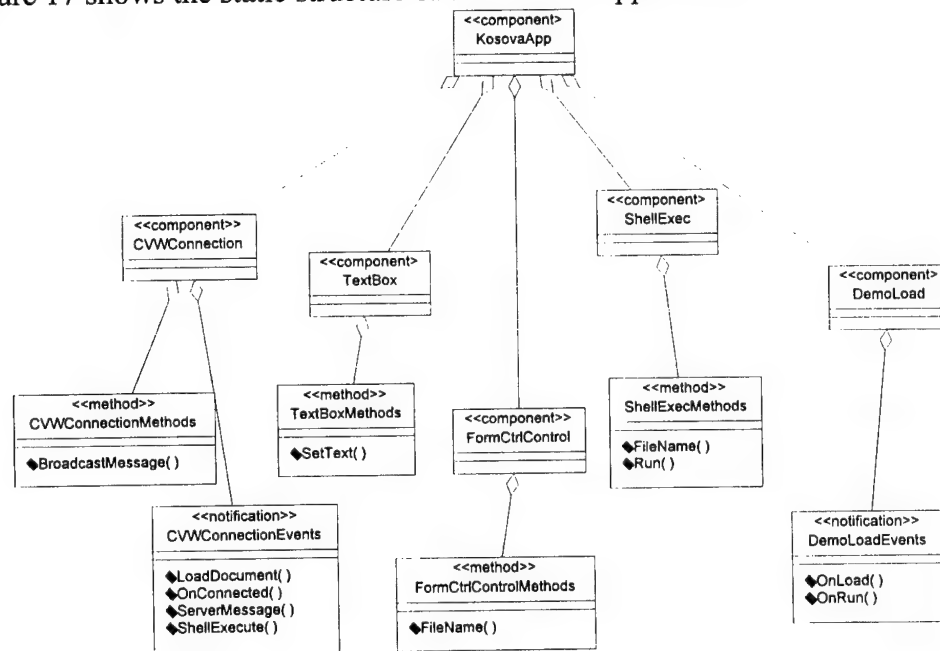


Figure 17. The Static Structure of Kosova Demonstraion

Figure 18 shows the behavior specification of the Kosova demonstration.

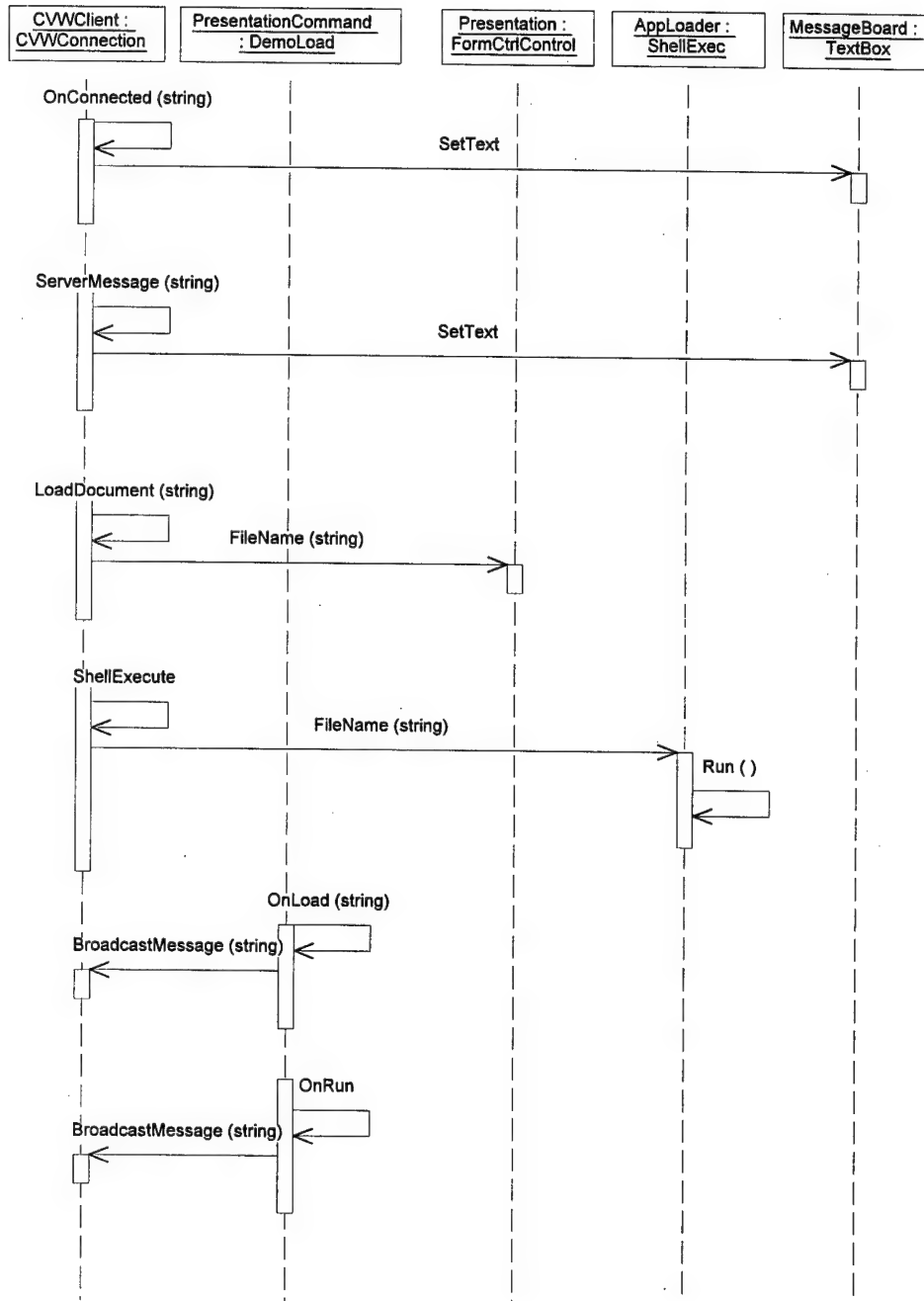


Figure 18. The Behavior Specification of KosovaApp

Figure 19 shows the static structure of a Kosova presentation.

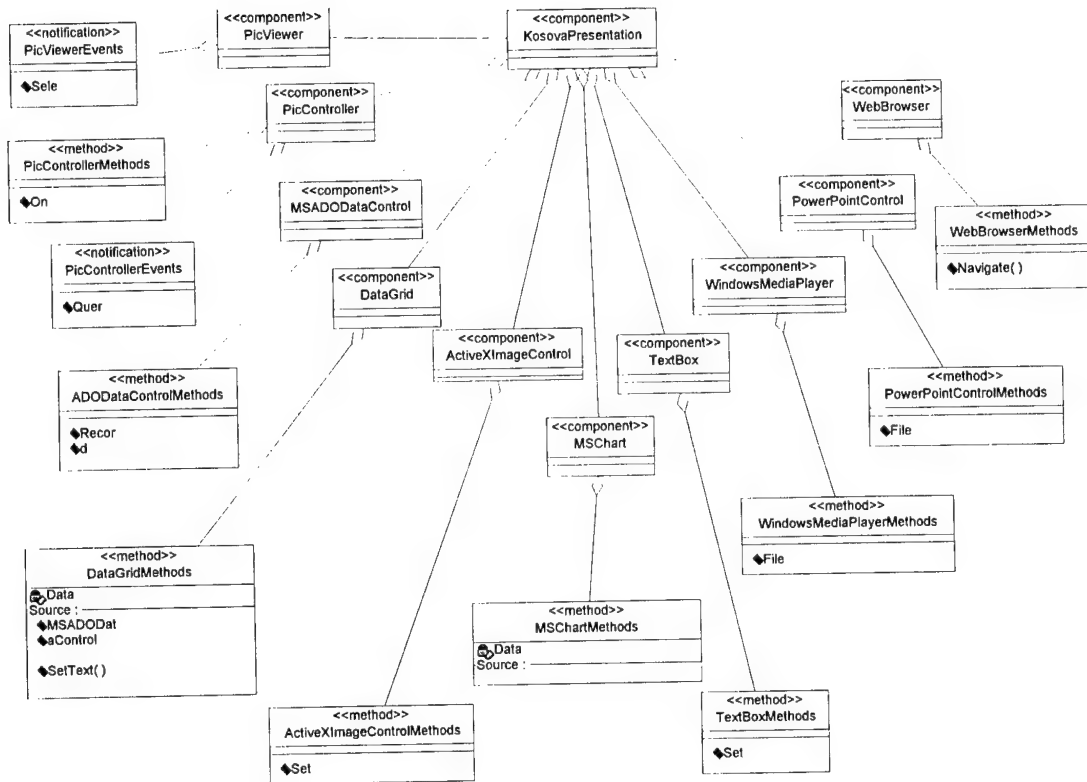


Figure 19. The Static Structure of a Presentation Component

Figures 20 and 21 show the behavior specification of a Kosova presentation.

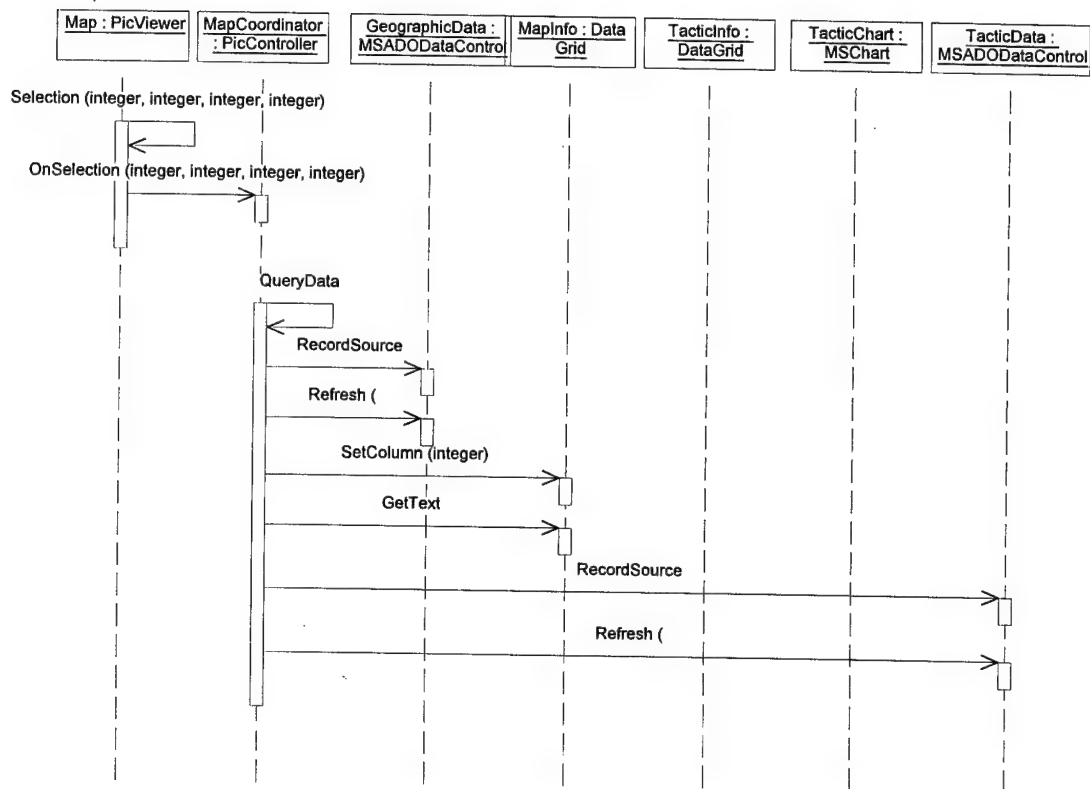


Figure 20. The behavior specification of the presentation in figure 3 (part 1)

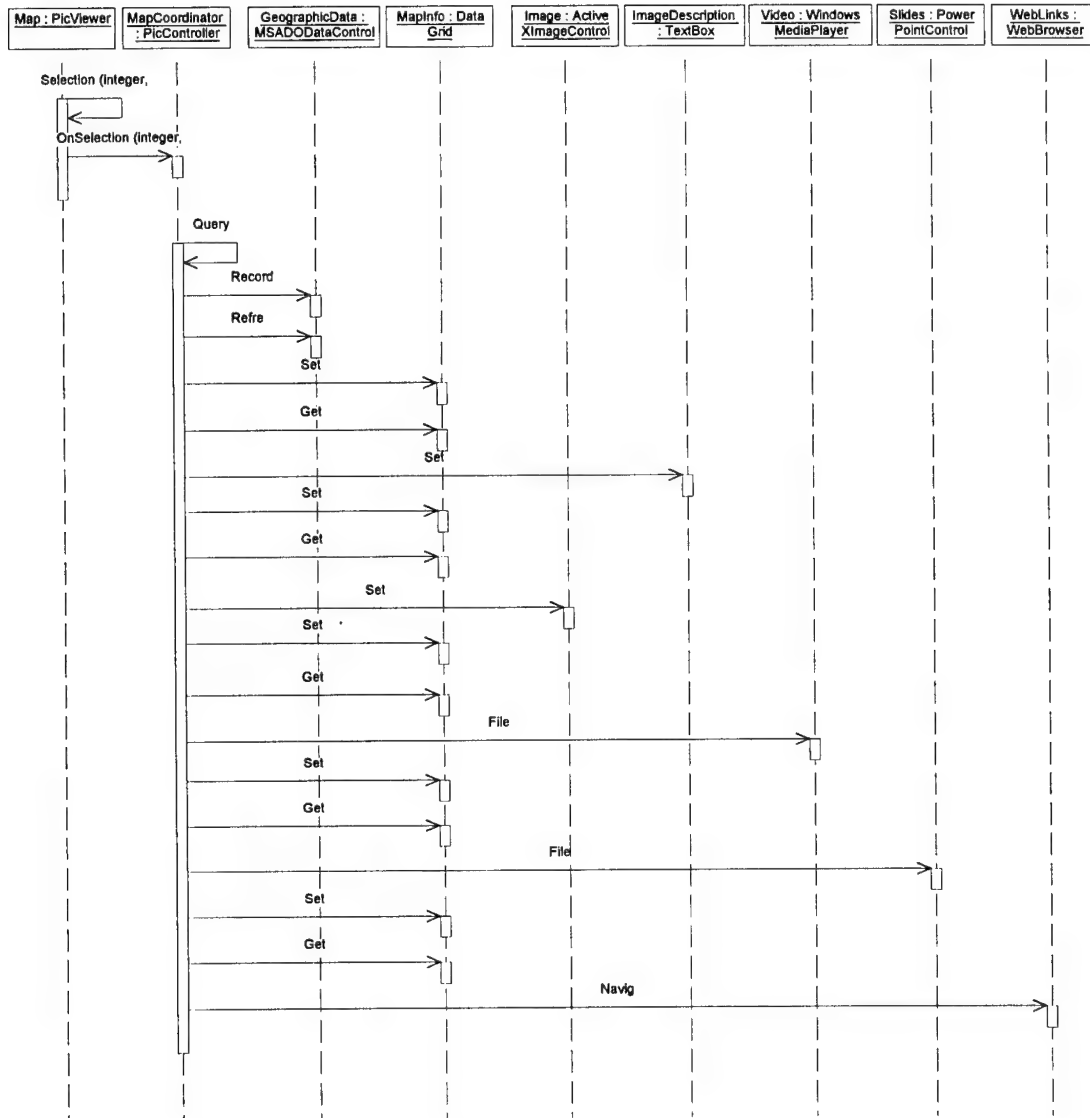


Figure 21. The behavior specification of the presentation in figure 3 (part 2)

Figure 22 shows the static structure of the CVWConnection.

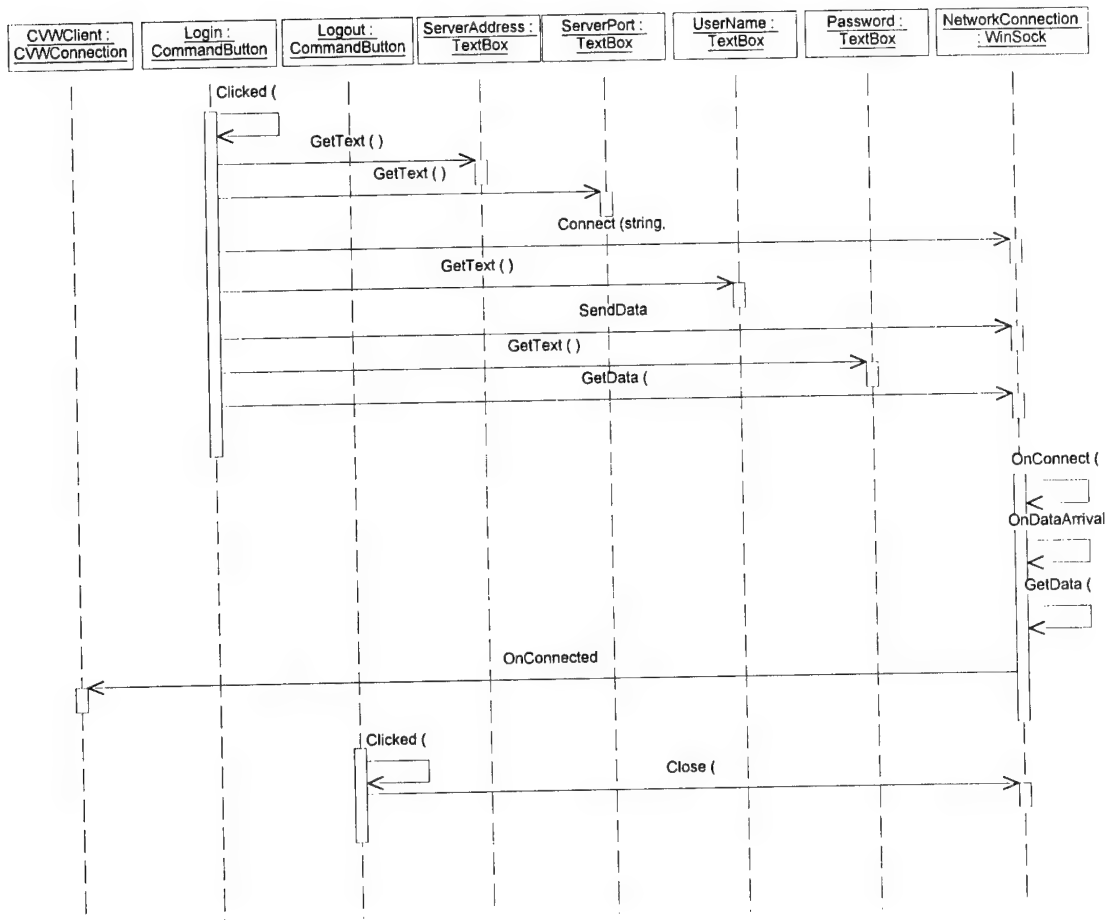


Figure 23. The behavior specification of the CVW Connection (part 1)

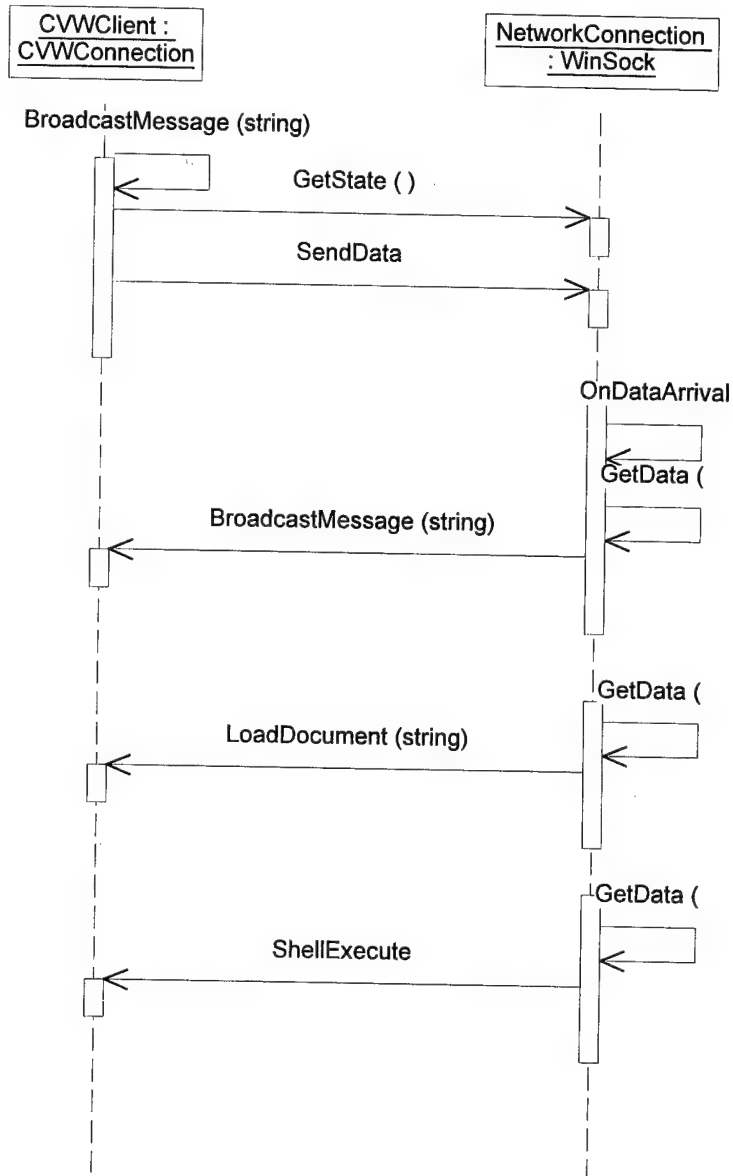


Figure 24. The behavior specification of the CVW Connection (part 2)

References

- [1]. N. Medvidovic and R. Taylor, "Separating Fact from Fiction in Software Architecture," ISAW '98: Proceedings of the Third International Workshop on Software Architecture, pp. 105-108, 1998.
- [2]. J. Ning, "CBSE Research at Andersen Consulting," Proceedings of 1998 International Workshop on Component-Based Software Engineering, 1998.
- [3]. B. Meyer, "Component and Object Technology: On to Components", IEEE Computer, Vol. 32, No. 1, pp. 139-140, January 1999.
- [4]. A. Iglesias and J. Justo, "Building System Requirements with Specification Components," Proceedings of Joint conference of Information and Computer Science, JICS'98, Vol. 3, pp. 499-502, October 1998.
- [5]. D. Garlan, R. Allen, and J. Ockerbloom, "Architectural Mismatch or Why it's Hard to Build Systems out of Existing Parts," Proceedings of the 17th International Conference on Software Engineering, pp. 179-185, April 1995.
- [6]. F. Achermann, and O. Nierstrasz, "Applications = Components + Scripts—A tour of Piccola," Software Architectures and Component Technology, Kluwer, 2000.
- [7]. M. Shaw and D. Garlan, "Software Architecture: Perspectives on an Emerging Discipline," Prentice Hall, April 1996.
- [8]. B. Johnson, "Letters: More Lessons from Civil Engineering," IEEE Software, Vol. 12, No. 6, pp. 6-6, November 1995.
- [9]. V. Baggiolini and J. Harms, "Toward Automatic, Run-Time Fault Management for Component-Based Applications," Object-Oriented Technology: ECOOP'97 Workshop Reader, Lecture Notes in Computer Science, Vol. 1357, pp. 327-331, Springer, 1997.
- [10]. C. Dellarocas, "Toward Exception Handling Infrastructures for Component-Based Software," Proceedings of 1998 International Workshop on Component-Based Software Engineering, 1998.
- [11]. UML RTF, "The Unified Modeling Language Version 1.3," The Object Management Group, 1999.

- [12]. M. Vaziri and D. Jackson, "Some Shortcomings of OCL, the Object Constraint Language of UML," Response to Object Management Group's Request for Information on UML 2.0, December 1999.
- [13]. M. Vaziri and D. Jackson, "Some Shortcomings of OCL, the Object Constraint Language of UML," Response to Object Management Group's Request for Information on UML 2.0, December 1999.
- [14]. Rational Software, "Rational Rose 2000e, Using Rose J," Revision 3.0, Rational Software Corporation, March 2000.
- [15]. Rational Software, "Rational Rose 2000e, Using Rose C++," Revision 7.0, Rational Software Corporation, April 2000.
- [16]. Rational Software, "Rational Rose 2000e, Using Rose Visual Basic," Revision 2.5, Rational Software Corporation, March 2000.
- [17]. G. Holzmann, "Design and Validation of Computer Protocols," Prentice-Hall, 1991.
- [18]. J. Salasin, "EDCS Program Overview," The Defense Advanced Research Projects Agency (DARPA), Information Technology Office (ITO), <http://www.darpa.mil/ito/research/edcs/reports-briefs.html>, June 1998.
- [19]. J. Hahn and J. Kim, "Why are Some Diagrams Easier to Work With? Effects of Diagrammatic Representation on the Cognitive Integration Process of Systems Analysis and Design", ACM Transactions on Computer Human Interaction, Vol. 6, No. 3, pp. 181-213, 1999.
- [20]. T. Reenskaug, "Working with Objects: The OOram Software Engineering Method," Manning Publications, 1996.

***MISSION
OF
AFRL/INFORMATION DIRECTORATE (IF)***

*The advancement and application of Information Systems Science
and Technology to meet Air Force unique requirements for
Information Dominance and its transition to aerospace systems to
meet Air Force needs.*